

Heuristic Algorithms for the Fixed-Charge Multiple Knapsack Problem

Byungjun You* Takeo Yamada†

Department of Computer Science, The National Defense Academy Yokosuka,
Kanagawa 239-8686, Japan

Abstract In a previous paper we formulated the fixed-charge multiple knapsack problem (FCMKP), and developed an exact algorithm to solve this problem to optimality by combining pegging test and branch-and-bound technique. Although the developed algorithm was quite successful for instances with small number of knapsacks (m), for larger m the problem was hard to solve by this method. Here we present some heuristic algorithms that can produce approximate solutions of satisfactory quality for FCMKPs with much larger m than in previous papers. We present greedy, local search and tabu search heuristics, and through numerical experiments evaluate these algorithms. As the result, we find that the local search method gives a satisfactory approximation for uncorrelated and weakly correlated cases, while in strongly correlated case tabu search is effective in obtaining solutions of high quality, although it is far more time consuming than other heuristics.

Keywords Integer programming; multiple knapsack problem; fixed-charge problem; tabu search.

1 Introduction

In a companion paper [8] we formulated the *fixed-charge multiple knapsack problem* (FCMKP) as an extension to the *multiple knapsack problem* (MKP), where we have n items $N := \{1, 2, \dots, n\}$ to be packed into m possible knapsacks $M := \{1, 2, \dots, m\}$. As in MKP, item j is associated with weight w_j and profit p_j ($j \in N$), and the capacity of knapsack i is c_i ($i \in M$). In FCMKP, however, a fixed cost f_i is charged if we use knapsack i , and thus the problem is to fill the knapsacks with items so that the total *net* profit is maximized, while the capacity constraints are all satisfied. FCMKP can be represented as a linear 0-1 programming problem using decision variables x_{ij} and y_i such that $x_{ij} = 1$ if item j is assigned to knapsack i , and $y_i = 1$ if we use knapsack i , respectively. The problem is formulated as follows.

FCMKP:

*Currently with the Republic of Korea Navy

†Corresponding author: E-mail: yamada.nda.ac.jp

$$\max \quad z(x, y) := \sum_{i=1}^m \sum_{j=1}^n p_j x_{ij} - \sum_{i=1}^m f_i y_i \quad (1)$$

$$\text{s. t.} \quad \sum_{j=1}^n w_j x_{ij} \leq c_i y_i, \quad i \in M \quad (2)$$

$$\sum_{i=1}^m x_{ij} \leq 1, \quad j \in N \quad (3)$$

$$x_{ij}, y_i \in \{0, 1\}, \quad i \in M, j \in N \quad (4)$$

Here, without much loss of generality we assume the following.

A₁: Problem data w_j , p_j ($j \in N$) and c_i , f_i ($i \in M$) are all positive integers.

A₂: Items are arranged in non-increasing order of profit per weight, i.e.,

$$p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n. \quad (5)$$

A₃: Knapsacks are numbered in non-increasing order of capacity per cost, i.e.,

$$c_1/f_1 \geq c_2/f_2 \geq \dots \geq c_m/f_m. \quad (6)$$

By $z(x, y)$ we denote the objective function for $x = (x_{ij})$, $y = (y_i)$, and (x^*, y^*) denotes an optimal solution with the corresponding objective value $z^* := z(x^*, y^*)$. FCMKP is \mathcal{NP} -hard [2], since the special case of zero knapsack costs ($f_i \equiv 0$ for all $i \in M$) is simply an MKP which is already \mathcal{NP} -hard.

Small instances of this problem may be solved using free or commercial MIP solvers [9]. Yasuda [10] gave a branch-and-price algorithm to solve FCMKP to optimality, and solved problems with up to $n \leq 200$ items. To solve larger instances, we developed an exact algorithm [8] that applies problem reduction by pegging test followed by branch-and-bound method with respect to decision variable y_i 's, where at each terminal subproblem we solve MKP using the C code developed by Pisinger [7]. For instances with $m \leq 50$ and $n \leq 32000$, the developed algorithm was quite successful. Indeed, it solved these FCMKPs exactly within a few seconds. However, for larger m the problem was hard to solve by our algorithm. This is especially the case when the ratio n/m is small, due to the weakness of the Pisinger's algorithm to such instances.

The aim of this article is to present some heuristic algorithms that can produce approximate solution of satisfactory quality for FCMKPs with m much larger than 50. The paper is organized as follows. In section 2, we apply the Lagrangian relaxation to FCMKP and derive an upper bound. We see that only a scalar Lagrangian variable suffices, and thus the Lagrangian upper bound can be obtained by a simple bisection method. Next, in section 3 three heuristic algorithms, namely *greedy*, *local search* and *tabu search* methods, are given to compute lower bounds. Finally, in section 4 we evaluate the developed algorithms through a series of numerical experiments on randomly generated instances of various statistical characteristics.

2 Lagrangian upper bound

This section derives an *upper bound* by applying the *Lagrangian relaxation* [1] to FCMKP. Details of the result of this section can be seen elsewhere [8]; nevertheless, we briefly repeat it here for reader's convenience.

With a nonnegative multiplier $\lambda = (\lambda_i) \in R^m$ associated with (2), the Lagrangian relaxation to FCMKP is

LFCMKP(λ):

$$\begin{aligned} \max \quad & \sum_{i=1}^m \sum_{j=1}^n (p_j - \lambda_i w_j) x_{ij} + \sum_{i=1}^m (\lambda_i c_i - f_i) y_i & (7) \\ \text{s. t.} \quad & (3), (4). \end{aligned}$$

For $\lambda \geq 0$, let $\bar{z}(\lambda)$ be the optimal objective value to LFCMKP(λ). Then, the following is immediate.

Theorem 1.

- (i) For an arbitrary $\lambda \geq 0$, $z^* \leq \bar{z}(\lambda)$; i.e., $\bar{z}(\lambda)$ gives an upper bound to FCMKP.
- (ii) $\bar{z}(\lambda)$ is a piecewise linear and convex function of λ .

Moreover, if we consider the *Lagrangian dual*

$$\min \bar{z}(\lambda) \quad \text{s. t.} \quad \lambda \geq 0,$$

we have the following [8].

Theorem 2. *There exists an optimal solution $\lambda^\dagger = (\lambda_i^\dagger)$ to the Lagrangian dual such that $\lambda_1^\dagger = \lambda_2^\dagger = \dots = \lambda_m^\dagger$.*

From this theorem, it suffices to consider the case of all $\lambda_i \equiv \lambda$, and LFCMKP(λ) is rewritten as the following problem with a single parameter λ .

LFCMKP(λ):

$$\max \quad \sum_{j=1}^n (p_j - \lambda w_j) x_j + \sum_{i=1}^m (\lambda c_i - f_i) y_i \quad (8)$$

$$\text{s. t.} \quad x_j, y_i \in \{0, 1\}, \quad j \in N, i \in M, \quad (9)$$

where we put

$$x_j := \sum_{i=1}^m x_{ij}.$$

The solution to this problem is given as

$$x_j(\lambda) = u(p_j - \lambda w_j), \quad y_i(\lambda) = u(c_i - f_i),$$

where $u(\cdot)$ is the *unit step function* defined by $u(s) = 1$ if $s \geq 0$, and $u(s) = 0$ otherwise.

Correspondingly, with $(\cdot)^+$ defined by $(s)^+ := \max\{s, 0\}$ the objective value is

$$\bar{z}(\lambda) = \sum_{j=1}^n (p_j - \lambda w_j)^+ + \sum_{i=1}^m (\lambda c_i - f_i)^+. \quad (10)$$

We note that

$$d\bar{z}(\lambda)/d\lambda = \sum_{\lambda c_i - f_i > 0} c_i - \sum_{p_j - \lambda w_j > 0} w_j, \quad (11)$$

provided that $\bar{z}(\lambda)$ is differentiable at λ . Then, the optimal λ^\dagger can be found by the *bisection method*, and we have the Lagrangian upper bound $\bar{z} := \bar{z}(\lambda^\dagger)$.

3 Lower bounds

3.1 A greedy method

For a standard 0-1 *knapsack problem* (KP) with knapsack capacity c and items being arranged according to (5), Pisinger [7] proposed the *forward* and *backward greedy* lower bounds as

$$\underline{z}_f = \max_{j=b, \dots, n} \{\bar{p}_{b-1} + p_j \mid \bar{w}_{b-1} + w_j \leq c\}, \quad (12)$$

and

$$\underline{z}_b = \max_{j=1, \dots, b-1} \{\bar{p}_b - p_j \mid \bar{w}_b - w_j \leq c\}, \quad (13)$$

where \bar{p}_j (\bar{w}_j) is the *accumulated profit* (and *weight* respectively), and b is the *critical item* which satisfies $\bar{w}_{b-1} < c \leq \bar{w}_b$. Then, $\max\{\underline{z}_f, \underline{z}_b\}$ gives a lower bound to KP.

We employ this method to solve FCMKP approximately but very quickly. For an arbitrary subset of items $I \subseteq N$, consider the knapsack problem with respect to knapsack i with capacity c_i . This is referred to as the knapsack problem $KP^i(I)$, and $N^i(I)$ denotes the set of accepted items by the above algorithm. The total profit (and weight) of $N^i(I)$ is denoted $\bar{p}^i(I)$ (and $\bar{w}^i(I)$). The set of items $N^i(I)$ is put into knapsack i if its profit $\bar{p}^i(I)$ is larger than the cost f_i . Then, our greedy algorithm for FCMKP is as follows. The output from this algorithm is denoted as (x_G, y_G) with the corresponding lower bound $\underline{z}_G := (x_G, y_G)$.

Algorithm GREEDY.

Step 1: (Initialization) Set $I := N$, and $i := 1$.

Step 2: (Check knapsack i) Apply Pisinger's greedy algorithm to $KP^i(I)$ and obtain $N^i(I)$.

Step 3: If $\bar{p}^i(I) < f_i$, go to Step 5.

Step 4: (Fill knapsack i) Let $I \leftarrow I \setminus N^i(I)$.

Step 5: If $i \geq n$ stop. Else $i \leftarrow i + 1$ and go to Step 2.

3.2 Local search

Let (x, y) be a feasible solution to FCMKP. We call item j *free* in this solution if it is not included in any knapsacks, i.e., $x_{ij} = 0$ for all $i \in M$. Similarly, knapsack i is free in (x, y) if $y_i = 0$.

For two feasible solutions (x, y) and (x', y') to FCMKP, we call the latter *1-opt neighbor* of the former if $(x, y) \equiv (x', y')$, except for some $j \in N$ and $i \in M$ item j is free in (x, y) and $x'_{ij} = 1$. That is, (x', y') is obtained from (x, y) by putting a free item j into knapsack i . We call (i, j) an *entering pair*, and by $N_1(x, y)$ denote the set of all 1-opt neighbors of (x, y) .

Next, (x', y') is said to be a *2-opt neighbor* of (x, y) , if for some $j \in N$, $i \in M$ and a non-empty set $N' \subseteq N$ of free items, (x', y') is obtained from (x, y) by replacing item j in knapsack i with N' . That is, $x_{ij} = 1$, $x_{hl} = 0$, $x'_{hj} = 0$ and $x'_{il} = 1$ for all $h \in M$ and $l \in N'$. We call (i, j) an *exiting pair*, and the set of all 2-opt neighbors of (x, y) is denoted by $N_2(x, y)$.

Another neighbor is obtained from (x, y) by adopting a free knapsack in (x, y) and filling it with a set $N' \subseteq N$ of free items. That is, $y_i = 0$, $x_{hl} = 0$, $y'_i = 1$ and $x'_{il} = 1$ for all $l \in N'$ and $h \in M$. $N_I(x, y)$ denotes the set of all these neighbors, and we call it the neighborhood by *knapsack augmentation*.

Finally, we introduce $N_D(x, y)$ as the neighborhood by *knapsack elimination*. This is the set of feasible solutions obtained from (x, y) by making a knapsack i free and moving all the items therein to other knapsacks (or making some of them free). That is, $y_i = 1$, $y'_i = 0$ and $x'_{il} = 0$ for all $l \in N$.

We note that all these relations are non-reflective, e.g., $(x', y') \in N_1(x, y)$ does not imply $(x, y) \in N_1(x', y')$. A solution (x', y') in either one these neighborhoods of (x, y) is said to be *improving* if $z(x, y) < z(x', y')$. Then, the local search algorithm for FCMKP is quite standard: we start with the greedy solution, and at each iteration look for an improving solution in the neighborhoods of the current solution. More precisely, we describe the algorithm as follows.

Algorithm LOCAL_SEARCH.

- Step 1:** (Initialization) Start with the greedy solution; i.e., set $(x, y) := (x_G, y_G)$.
Step 2: If there exists an improving solution (x', y') either in $N_2(x, y)$, $N_1(x, y)$, $N_D(x, y)$ or $N_I(x, y)$, go to Step 4.
Step 3: (local maximal) Output (x, y) as the local search solution and stop.
Step 4: Update solution as $(x, y) \leftarrow (x', y')$ and go to Step 2.

The output from this algorithm is the local search solution (x_L, y_L) with the corresponding lower bound $\underline{z}_L := z(x_L, y_L)$. The order by which neighborhoods are scanned in Step 2, i.e., in this case $N_2(x, y) \rightarrow N_1(x, y) \rightarrow N_D(x, y) \rightarrow N_I(x, y)$, may be of critical importance to the efficiency of the algorithm. In FCMKP we expect that most improvements will occur as a result of a movement from current (x, y) to some (x', y') in the 2-opt neighborhood. Improvements involving other neighborhoods are considered to occur less frequently. Therefore, we first examine $N_2(x, y)$, and then $N_1(x, y)$ which is easier than other neighborhoods to explore.

3.3 Tabu search

It is easily expected that in the local search method a local maximum is reached in relatively small number of steps, and we often end up with a solution of unsatisfactory quality. To overcome this weakness, in tabu search we do not stop even if we reach local maximum. Here, instead of stopping at Step 3 of LOCAL_SEARCH we move to the ‘least worsening’ solution in $N_2(x,y)$ and go back to Step 2. Then, to prevent an obvious cycling such as $(x,y) \rightarrow (x',y') \rightarrow (x,y) \rightarrow \dots$, we prepare a *tabu list* \mathcal{T} and some appropriate stopping rule in the following.

The output from this algorithm is the tabu search solution (x_T, y_T) with the corresponding lower bound $z_T := z(x_T, y_T)$. To implement the following TABU_SEARCH, it is necessary to specify the details of the algorithm, including the structure of the tabu list \mathcal{T} , the way the list is handled, and the stopping rule of computation. In this paper, \mathcal{T} is a *queue* of size TL (tabu length), consisting of tabu entities of the form $(i, j) \in N \times M$. As a queue, \mathcal{T} works in *first-in, first-out* (FIFO) rule, and each time we execute Step 3, we put the exiting pair (i, j) into the tabu list \mathcal{T} . If $|\mathcal{T}| > \text{TL}$, the oldest member of \mathcal{T} is eliminated from the list.

Here we have a choice on the structure of the tabu entities. As stated above, this may take the form of a pair $(i, j) \in N \times M$, which we call Strategy A1. Alternatively, in Strategy A2 we only keep the history of items. That is, in this case the tabu list is simply a subset of items, i.e., $\mathcal{T} \subseteq N$, and we only check if the item being considered is included in the tabu list, irrespective to knapsacks.

Algorithm TABU_SEARCH.

- Step 1:** (Initialization) Start with the local search solution; i.e., set $(x,y) := (x_L, y_L)$. This is also the *incumbent* solution (x_I, y_I) with the incumbent objective value $z_I := z(x_I, y_I)$.
- Step 2:** (2-opt) If there exists an improving solution (x', y') either in $N_2(x,y) \setminus \mathcal{T}$, $N_1(x,y)$, $N_D(x,y)$ or $N_I(x,y)$, go to Step 4.
- Step 3:** (Local maximum) Let (x', y') be the least worsening solution in $N_2(x,y) \setminus \mathcal{T}$, and (i, j) be an exiting pair in 2-opt move from (x,y) to (x', y') . Then, update the tabu list by setting $\mathcal{T} \leftarrow \mathcal{T} \cup \{(i, j)\}$.
- Step 4:** Let $(x,y) \leftarrow (x', y')$. If (x,y) is better than the incumbent, i.e., $z(x,y) > z_I$, update the incumbent as $(x_I, y_I) \leftarrow (x,y)$ and $z_I \leftarrow z(x,y)$.
- Step 5:** If stopping rule is satisfied, output incumbent solution and stop. Otherwise, go back to Step 2.

Next, in Step 2 we look for a set of free items that can be put into knapsack i in place of item j . We examine if the exiting pair is tabooed, i.e., if $(i, j) \in \mathcal{T}$, and if this is the case (i, j) is skipped from further consideration. We may also examine the entering items by checking if $(i, l) \in \mathcal{T}$ for free item l . Then, if we only examine exiting pairs, we call this Strategy B1. In Strategy B2, both entering and exiting pairs are tabu-checked.

Finally, we discuss the stopping rule for TABU_SEARCH. By MAXIT we specify the maximum number of successive execution of Step 3 (non-improving move) without improving incumbent lower bound. As soon as MAXIT executions of Step 3 are repeated without improving the incumbent lower bound, the process is terminated.

4 Numerical experiments

In this section we first tune up the `TABU_SEARCH` parameters, and then analyze the heuristic algorithms of the previous sections through a series of numerical experiments. We implemented the algorithm in ANSI C language and conducted computation on an DELL Precision 670 computer (CPU: Intel Xeon 3.20GHz).

4.1 Design of experiment

The size of the instances tested is $n = 1000 \sim 32000$ and $m = 100 \sim 500$, and instances are prepared according to the following scheme [8].

(a) Items

- weight w_j : Uniformly random integer over $[1, 1000]$.
- profit p_j
 - Uncorrelated (UNCOR): Uniformly random over $[1, 1000]$, independent of w_j .
 - Weakly correlated (WEAK): Uniformly random over $[w_j, w_j + 200]$.
 - Strongly correlated (STRONG): $p_j := w_j + 20$.

(b) Knapsacks

- Capacity $c_i = \lfloor 500n \cdot \delta \cdot \xi_i \rfloor$. Here ξ_i ($i \in M$) is uniformly distributed over $\{(\xi_1, \dots, \xi_m) \mid \sum_{i=1}^m \xi_i = 1, \xi_i \geq 0\}$ and δ is either 0.25, 0.50 or 0.75.
- Cost $f_i = \rho_i c_i$, where ρ_i is uniformly random over $[0.5, 1.5]$.

Here the parameter δ in the knapsack capacity controls the ratio of items that can be accepted into the knapsacks. Since the average weight of items is approximately 500, $\delta = 0.50$ means that about a half of all the items can be accommodated in the knapsacks.

4.2 Parameter tuning for `TABU_SEARCH`

Table 1 compares the effects of tabu length (TL) on the quality of the solution (Error%) and CPU time in seconds (CPU) for the cases of $n = 2000, m = 300$ and $n = 8000, m = 500$ under various correlation levels between weights and profits of items. Here, Error% is the *relative error* of the tabu search solution against the upper bound, i.e.,

$$\text{Error\%} := 100.0 \times (\bar{z} - \underline{z}_{TS}) / \bar{z}_{TS} \quad (14)$$

and in Table 1 (and in subsequent tables) each row is the average over 10 randomly generated instances. Considering both Error% and CPU in Table 1, we determine the tabu length at TL= 20. Table 2 summarizes the effect of the stopping parameter MAXIT. Again, considering both Error% and CPU in Table 2 we determine this parameter as MAXIT = 50.

4.3 Analysis of `TABU_SEARCH`

Tables 3 and 4 compare strategies A1 vs. A2 and B1 vs. B2, respectively. From these tables, we see that these strategies do not significantly affect the performance of `TABU_SEARCH`. Thus, in what follows we employ strategies A1 and B1 as standard.

Table 1: Parameter tuning: TL

| n | m | TL | UNCOR | | WEAK | | STRONG | |
|------|-----|----|--------|------|--------|------|--------|-------|
| | | | Error% | CPU | Error% | CPU | Error% | CPU |
| 2000 | 300 | 10 | 1.25 | 0.71 | 1.81 | 0.85 | 3.19 | 1.61 |
| | | 20 | 1.24 | 0.76 | 1.81 | 1.00 | 2.45 | 1.81 |
| | | 30 | 1.25 | 0.64 | 1.79 | 0.82 | 2.64 | 1.97 |
| | | 40 | 1.25 | 0.73 | 1.77 | 0.90 | 2.54 | 1.99 |
| 8000 | 500 | 10 | 0.31 | 7.22 | 0.44 | 8.24 | 1.42 | 28.64 |
| | | 20 | 0.31 | 7.44 | 0.44 | 6.91 | 1.19 | 34.72 |
| | | 30 | 0.31 | 6.81 | 0.44 | 7.37 | 1.17 | 35.07 |
| | | 40 | 0.31 | 6.74 | 0.44 | 6.91 | 1.11 | 33.95 |

Table 2: Parameter tuning: MAXIT

| n | m | MAXIT | UNCOR | | WEAK | | STRONG | |
|------|-----|-------|--------|-------|--------|-------|--------|-------|
| | | | Error% | CPU | Error% | CPU | Error% | CPU |
| 2000 | 300 | 30 | 1.25 | 0.48 | 1.82 | 0.51 | 2.45 | 1.52 |
| | | 50 | 1.24 | 0.76 | 1.81 | 1.00 | 2.45 | 1.81 |
| | | 70 | 1.24 | 1.33 | 1.80 | 1.14 | 2.45 | 1.95 |
| | | 90 | 1.23 | 1.93 | 1.79 | 1.43 | 2.44 | 2.14 |
| 8000 | 500 | 30 | 0.31 | 4.93 | 0.44 | 5.24 | 1.25 | 29.87 |
| | | 50 | 0.31 | 7.44 | 0.44 | 6.91 | 1.19 | 34.72 |
| | | 70 | 0.31 | 9.80 | 0.43 | 10.67 | 1.19 | 36.43 |
| | | 90 | 0.31 | 12.45 | 0.43 | 12.53 | 1.19 | 37.56 |

Next, we show in Table 5 the result of sensitivity analysis on the capacity parameter δ . We observe here that we usually obtain better solutions for larger δ , although in CPU time differences are not significant.

Table 6 gives a summary of the neighborhood behaviors in TABU_SEARCH, where total numbers of neighborhood operations (#Iter) and percentages of four neighborhood operations are shown. The number of operations increases with the size of instances, and most of the tabu search movements are carried out through 2-opt operations. Incr and 1-opt operations contribute to some extent, but Decr is seldom used in the algorithm.

4.4 Performance evaluation

Finally, Tables 7-9 give a comprehensive summary of the performance of the heuristic algorithms. Here shown are the solution quality (Error%) and computing time (CPU) of GREEDY, LOCAL_SEARCH and TABU_SEARCH methods for various sizes of instances with different correlation levels. From these tables we observe the followings.

- In UNCOR and WEAK instances, the quality of these solutions is quite satisfactory with the relative error being less than a few percent except for some cases with $n \leq 2000$. In these cases, TABU_SEARCH gives slightly better solutions in expense of much longer computing time.
- Advantage of TABU_SEARCH over GREEDY and LOCAL_SEARCH in solution quality is clear in STRONG cases, although it is far more time-consuming to compute.
- In TABU_SEARCH, the solution quality heavily depends on the ratio n/m . For all correlation types, we have solutions of poor quality for small n and large m , such as $n = 1000$ and $m = 500$.

Table 3: Strategy A compared

| n | m | CORR | Strategy A1 | | Strategy A2 | |
|------|-----|--------|-------------|-------|-------------|-------|
| | | | Error% | CPU | Error% | CPU |
| 2000 | 300 | UNCOR | 1.24 | 0.76 | 1.29 | 0.70 |
| | | WEAK | 1.81 | 1.00 | 1.90 | 0.67 |
| | | STRONG | 2.45 | 1.81 | 2.77 | 3.61 |
| 8000 | 500 | UNCOR | 0.31 | 7.44 | 0.32 | 5.68 |
| | | WEAK | 0.44 | 6.91 | 0.46 | 7.38 |
| | | STRONG | 1.19 | 34.72 | 1.19 | 46.25 |

Table 4: Strategy B compared

| n | m | CORR | Strategy B1 | | Strategy B2 | |
|------|-----|--------|-------------|-------|-------------|-------|
| | | | Error% | CPU | Error% | CPU |
| 2000 | 300 | UNCOR | 1.24 | 0.76 | 1.27 | 0.59 |
| | | WEAK | 1.81 | 1.00 | 1.82 | 1.00 |
| | | STRONG | 2.45 | 1.81 | 2.38 | 1.81 |
| 8000 | 500 | UNCOR | 0.31 | 7.44 | 0.31 | 6.98 |
| | | WEAK | 0.44 | 6.91 | 0.44 | 7.24 |
| | | STRONG | 1.19 | 34.72 | 1.20 | 34.07 |

Table 5: Sensitivity analysis on δ

| n | m | CORR | $\delta = 0.25$ | | $\delta = 0.50$ | | $\delta = 0.75$ | |
|------|-----|--------|-----------------|-------|-----------------|-------|-----------------|-------|
| | | | Error% | CPU | Error% | CPU | Error% | CPU |
| 2000 | 300 | UNCOR | 2.31 | 1.26 | 1.24 | 0.76 | 0.88 | 0.59 |
| | | WEAK | 3.09 | 1.17 | 1.81 | 1.00 | 1.21 | 0.86 |
| | | STRONG | 4.28 | 2.06 | 2.45 | 1.81 | 2.13 | 2.00 |
| 8000 | 500 | UNCOR | 0.62 | 8.27 | 0.31 | 7.44 | 0.21 | 5.81 |
| | | WEAK | 0.77 | 8.11 | 0.44 | 6.91 | 0.30 | 8.37 |
| | | STRONG | 1.54 | 32.07 | 1.19 | 34.72 | 0.98 | 35.43 |

Table 6: TABU_SEARCH operations

| n | m | CORR | #Iter | 1-opt(%) | 2-opt(%) | Decr(%) | Incr(%) |
|------|-----|--------|-------|----------|----------|---------|---------|
| 2000 | 300 | UNCOR | 138.7 | 9.30 | 85.87 | 0.87 | 3.97 |
| | | WEAK | 309.5 | 0.23 | 95.57 | 0.35 | 3.84 |
| | | STRONG | 330.2 | 6.78 | 78.56 | 1.79 | 12.87 |
| 8000 | 500 | UNCOR | 175.3 | 6.50 | 90.36 | 0.23 | 2.74 |
| | | WEAK | 502.7 | 0.00 | 98.23 | 0.08 | 1.69 |
| | | STRONG | 534.8 | 4.94 | 85.81 | 1.12 | 8.13 |

Table 7: Performance of heuristic algorithms (UNCOR)

| n | m | GREEDY | | LOCAL_SEARCH | | TABU_SEARCH | |
|-------|-----|--------|------|--------------|------|-------------|-------|
| | | Error% | CPU | Error% | CPU | Error% | CPU |
| 1000 | 100 | 1.06 | 0.00 | 0.95 | 0.00 | 0.87 | 0.11 |
| | 300 | 3.95 | 0.00 | 3.49 | 0.01 | 3.21 | 0.38 |
| | 500 | 7.09 | 0.00 | 6.45 | 0.01 | 6.00 | 0.92 |
| 2000 | 100 | 0.39 | 0.00 | 0.35 | 0.00 | 0.33 | 0.30 |
| | 300 | 1.55 | 0.01 | 1.36 | 0.01 | 1.24 | 0.79 |
| | 500 | 2.85 | 0.01 | 2.53 | 0.01 | 2.28 | 1.70 |
| 4000 | 100 | 0.15 | 0.00 | 0.13 | 0.02 | 0.13 | 1.03 |
| | 300 | 0.56 | 0.00 | 0.49 | 0.02 | 0.45 | 1.78 |
| | 500 | 1.06 | 0.00 | 0.93 | 0.04 | 0.87 | 2.77 |
| 8000 | 100 | 0.05 | 0.00 | 0.04 | 0.07 | 0.04 | 3.76 |
| | 300 | 0.20 | 0.01 | 0.17 | 0.08 | 0.16 | 5.09 |
| | 500 | 0.38 | 0.02 | 0.33 | 0.09 | 0.31 | 7.77 |
| 16000 | 100 | 0.02 | 0.01 | 0.02 | 0.27 | 0.02 | 15.03 |
| | 300 | 0.07 | 0.02 | 0.06 | 0.30 | 0.06 | 18.16 |
| | 500 | 0.15 | 0.03 | 0.12 | 0.32 | 0.12 | 20.96 |
| 32000 | 100 | 0.01 | 0.01 | 0.01 | 1.06 | 0.01 | 57.37 |
| | 300 | 0.02 | 0.03 | 0.02 | 1.45 | 0.02 | 83.04 |
| | 500 | 0.05 | 0.07 | 0.04 | 1.12 | 0.04 | 71.71 |

Table 8: Performance of heuristic algorithms (WEAK)

| n | m | GREEDY | | LOCAL_SEARCH | | TABU_SEARCH | |
|-------|-----|--------|------|--------------|------|-------------|-------|
| | | Error% | CPU | Error% | CPU | Error% | CPU |
| 1000 | 100 | 2.66 | 0.00 | 1.57 | 0.00 | 1.31 | 0.12 |
| | 300 | 10.71 | 0.00 | 7.37 | 0.00 | 5.18 | 0.60 |
| | 500 | 18.47 | 0.00 | 13.90 | 0.01 | 9.97 | 1.60 |
| 2000 | 100 | 0.94 | 0.00 | 0.50 | 0.00 | 0.49 | 0.39 |
| | 300 | 3.95 | 0.00 | 2.36 | 0.01 | 1.81 | 1.17 |
| | 500 | 6.99 | 0.00 | 4.68 | 0.01 | 3.24 | 2.18 |
| 4000 | 100 | 0.34 | 0.00 | 0.20 | 0.02 | 0.18 | 0.19 |
| | 300 | 1.44 | 0.00 | 0.77 | 0.02 | 0.65 | 2.07 |
| | 500 | 2.67 | 0.00 | 1.57 | 0.03 | 1.20 | 3.13 |
| 8000 | 100 | 0.12 | 0.00 | 0.06 | 0.07 | 0.06 | 4.27 |
| | 300 | 0.51 | 0.01 | 0.26 | 0.08 | 0.23 | 6.03 |
| | 500 | 0.96 | 0.01 | 0.52 | 0.09 | 0.44 | 7.24 |
| 16000 | 100 | 0.04 | 0.00 | 0.02 | 0.28 | 0.02 | 15.24 |
| | 300 | 0.18 | 0.01 | 0.09 | 0.30 | 0.08 | 19.55 |
| | 500 | 0.37 | 0.02 | 0.18 | 0.32 | 0.16 | 23.67 |
| 32000 | 100 | 0.01 | 0.01 | 0.01 | 1.13 | 0.01 | 67.72 |
| | 300 | 0.06 | 0.04 | 0.03 | 1.54 | 0.03 | 88.47 |
| | 500 | 0.12 | 0.07 | 0.06 | 1.27 | 0.05 | 71.50 |

Table 9: Performance of heuristic algorithms (STRONG)

| n | m | GREEDY | | LOCAL_SEARCH | | TABU_SEARCH | |
|-------|-----|--------|------|--------------|------|-------------|--------|
| | | Error% | CPU | Error% | CPU | Error% | CPU |
| 1000 | 100 | 20.55 | 0.00 | 17.33 | 0.00 | 2.24 | 0.24 |
| | 300 | 55.53 | 0.00 | 45.30 | 0.00 | 6.00 | 0.90 |
| | 500 | 95.17 | 0.00 | 70.46 | 0.00 | 13.80 | 1.88 |
| 2000 | 100 | 10.94 | 0.00 | 9.38 | 0.00 | 1.49 | 0.72 |
| | 300 | 31.28 | 0.00 | 26.34 | 0.01 | 2.45 | 1.80 |
| | 500 | 49.61 | 0.00 | 40.11 | 0.01 | 4.70 | 4.12 |
| 4000 | 100 | 5.59 | 0.00 | 4.90 | 0.02 | 0.61 | 2.76 |
| | 300 | 15.51 | 0.00 | 13.97 | 0.02 | 1.29 | 5.75 |
| | 500 | 25.64 | 0.00 | 22.25 | 0.03 | 2.39 | 10.15 |
| 8000 | 100 | 2.91 | 0.00 | 2.60 | 0.07 | 0.33 | 10.29 |
| | 300 | 7.90 | 0.01 | 7.18 | 0.09 | 0.77 | 21.08 |
| | 500 | 13.12 | 0.01 | 11.98 | 0.09 | 1.19 | 34.46 |
| 16000 | 100 | 1.47 | 0.00 | 1.33 | 0.28 | 0.20 | 32.64 |
| | 300 | 4.09 | 0.02 | 3.80 | 0.29 | 0.48 | 56.88 |
| | 500 | 6.60 | 0.03 | 6.18 | 0.32 | 0.71 | 88.91 |
| 32000 | 100 | 0.77 | 0.02 | 0.68 | 1.40 | 0.11 | 152.54 |
| | 300 | 2.11 | 0.04 | 1.99 | 1.22 | 0.32 | 215.87 |
| | 500 | 3.48 | 0.06 | 3.32 | 1.22 | 0.40 | 322.89 |

5 Conclusion

We have formulated the fixed-charge multiple knapsack problem, and presented some heuristic algorithms to solve this problem to approximately. The greedy and local search methods gave satisfactory solutions for uncorrelated and weakly correlated cases; while the tabu search algorithm was superior in strongly correlated case, with the expense of much longer CPU time.

References

- [1] M. Fisher, "The Lagrangian relaxation method for solving integer programming problems," *Management Science* **50** (2004), 1861-1871.
- [2] M.R. Garey, D.S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman and Company, San Francisco, 1979.
- [3] H. Kellerer, U. Pferschy, D. Pisinger: *Knapsack Problems*, Springer, 2004.
- [4] S. Martello and P. Toth: *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley and Sons, New York, 1990.
- [5] G.L. Nemhauser, L.A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley and Sons, New York, 1988.
- [6] D. Pisinger: "An expanding-core algorithm for the exact 0-1 knapsack problem," *European Journal of Operational Research* **87**(1995), 175-187.
- [7] D. Pisinger: "An exact algorithm for large multiple knapsack problems," *European Journal of Operational Research* **114**(1990), 528-541. (Source code 'mulknaps' available at <http://www.diku.dk/pisinger/codes.html>)

- [8] T. Takeoka, T. Yamada: "An Exact Algorithm for the Fixed-Charge Multiple Knapsack Problem", to appear in *European Journal of Operational Research*.
- [9] XPRESS-IVE Ver. 1.16.20: Dash Associates, 2006 (<http://www.dashoptimization.com>)
- [10] R. Yasuda, "An algorithm for the Fixed-charge multiple knapsack problem" (in Japanese), unpublished Master's Thesis, National Defense Academy, 2005.